The Little Manual of API Design

Jasmin Blanchette
Trolltech, a Nokia company

June 19, 2008

Contents

Chapter 1

Introduction

An application programming interface, or API, is the set of symbols that are exported and available to the users of a library to write their applications. The design of the APIs is arguably the most critical part of designing the library, because it affects the design of the applications built on top of them. To quote Daniel Jackson[1]:

> Software is built on abstractions. Pick the right ones, and programming will flow naturally from design; modules will have small and simple interfaces; and new functionality will more likely fit in without extensive reorganization. Pick the wrong ones, and programming will be a series of nasty surprises.

This manual gathers together the key insights into API design that were discovered through many years of software development on the Qt application development framework at Trolltech (now part of Nokia). When designing and implementing a library, you should also keep other factors in mind, such as efficiency and ease of implementation, in addition to pure API considerations. And although the focus is on public APIs, there is no harm in applying the principles described here when writing application code or internal library code.

Qt! Examples from Qt's history are presented in blocks like this one. If you are new to Qt, you might find some of these examples obscure. Don't hesitate to ask your colleagues for details. Also, many of the examples come from classes on which I worked, for the simple reason that I know those best. Other classes could have provided just as good examples.

Acknowledgment. My first acknowledgment is of Lars Knoll, who encouraged me to write this manual and gave feedback during the writing. I would also like to thank Frans Englich, Andreas Aardal Hanssen, Simon

[1] Daniel Jackson, *Software Abstractions*, MIT Press, 2006.

Hausmann, Mark Summerfield, Jan-Arve Sæther, and Trenton W. Schulz for reviewing manuscripts of this manual, and all the Trolltech developers with whom I have had fruitful API discussions.

Chapter 2

Characteristics of Good APIs

What is a good API? Although the concept is inherently subjective, most library developers seem to agree on the main desirable characteristics of an API. The list below is inspired by a talk by Joshua Bloch[1] and an article by Matthias Ettrich[2]:

- **Easy to learn and memorize**
- **Leads to readable code**
- **Hard to misuse**
- **Easy to extend**
- **Complete**

Observe that "minimal" and "consistent" don't appear on the list. They don't need to; a bloated, inconsistent API will be hard to learn and memorize, and also hard to extend. We should strive for minimality and consistency, but only to the extent that they contribute to meeting the criteria listed above.

"Consistency" coincides broadly with "conceptual integrity", the principle that a complex system should have a coherent design, reflecting the vision of one architect. Decades ago, Frederick Brooke wrote[3]:

> I contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas.

[1] Joshua Bloch, "How to Design a Good API and Why It Matters", Google Tech Talk, 2007. Available at `http://video.google.com/videoplay?docid=-3733345136856180693`.

[2] Matthias Ettrich, "Designing Qt-Style C++ APIs", *Qt Quarterly* 13, 2004. Available at `http://doc.trolltech.com/qq/qq13-apis.html`.

[3] Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1975. Second edition 1995.

Incidentally, if you are lucky enough to be extending a library that already exhibits the virtues listed above, mimicry is your fast track to success.

2.1 Easy to learn and memorize

An easy-to-learn API features consistent naming conventions and patterns, economy of concepts, and predictability. It uses the same name for the same concept, and different names for different concepts.

A minimal API is easy to memorize because there is little to remember. A consistent API is easy to memorize because you can reapply what you learned in one part of the API when using a different part.

An API is not only the names of the classes and methods that compose it, but also their intended semantics. The semantics should be simple and clear, and follow the principle of least surprise.

Some APIs are difficult to learn because they require the user to write heaps of boilerplate code just to get started. Nothing is more intimidating to the novice user than creating their first widget, 3D scene, or data model; and nothing is more destructive to their self-confidence than failing at that. An easy-to-learn API makes it possible to write the "hello world" example in just a few easy lines of code and to expand it incrementally to obtain more complex programs.

Qt! `QPushButton` is an example of a low-threshold class. You simply instantiate it and call `show()` on it, and behold, you have a `QPushButton` with its own window frame and taskbar entry. In retrospect, it would be even better if the call to `show()` could be omitted (e.g., by showing the widget in the next iteration of the event loop), and if the `QApplication` object was implicitly created by Qt. "Hello world" would then be

```
#include <QtGui>

int main()
{
    QPushButton button("Hello world");
    return QApplication::run();
}
```

Convenience methods—methods that are (or could be) implemented in terms of other public methods to make certain idioms shorter to write—contribute to API bloat. However, they are acceptable and indeed encouraged so long as they are clearly documented as "convenience" and fit nicely with the rest of the API. For example, if you provide an `insertItem(int, Item)` method that takes an index and an item, it often makes sense to also provide `addItem(Item)` as a convenience.

2.2 Leads to readable code

A programming language or library can be easy to learn and memorize, and yet lead to totally opaque code. The language APL is an extreme case; the expression $(\tilde{}R \in R \circ . \times R)/R \leftarrow 1 \downarrow \iota R$ returns a list of prime numbers up to R.[4] Closer to us, Perl acquired the reputation of being a "write-only language" because of its cryptic syntax.

Application code is written only once, but read over and over again by different developers over the lifetime of the application. Readable code is easier to document and maintain. It is also less likely to contain bugs, because the bugs are made more visible by the code's readability.

Qt! In Qt 3, `QSlider`'s constructor let you specify various properties:

```
slider = new QSlider(8, 128, 1, 6, Qt::Vertical, 0,
                     "volume");
```

In Qt 4, you would write

```
slider = new QSlider(Qt::Vertical);
slider->setRange(8, 128);
slider->setValue(6);
slider->setObjectName("volume");
```

which is easier to read, and even easier to write with no documentation or autocompletion. It is also easier to see that the value of 6 lies outside the range [8, 128].

Readable code can be concise or verbose. Either way, it is always at the right level of abstraction—neither hiding important things nor forcing the programmer to specify irrelevant information.

Qt! The Qt Jambi code

```
QGridLayout layout = new QGridLayout;
layout.addWidget(slider, 0, 0);
layout.addWidget(spinBox, 0, 1);
layout.addWidget(resetButton, 2, 1);
layout.setRowStretch(1, 1);
setLayout(layout);
```

is more readable than the equivalent Swing code:

```
GridBagLayout layout = new GridBagLayout();
GridBagConstraints constraint = new GridBagConstraints();

constraint.fill = GridBagConstraints.HORIZONTAL;
constraint.insets = new Insets(10, 10, 10, 0);
constraint.weightx = 1;
```

[4] Wikipedia, "APL programming language". Available at `http://en.wikipedia.org/wiki/APL_programming_language`.

```
layout.setConstraints(slider, constraint);

constraint.gridwidth = GridBagConstraints.REMAINDER;
constraint.insets = new Insets(10, 5, 10, 10);
constraint.weightx = 0;
layout.setConstraints(spinner, constraint);

constraint.anchor = GridBagConstraints.SOUTHEAST;
constraint.fill = GridBagConstraints.REMAINDER;
constraint.insets = new Insets(10, 10, 10, 10);
constraint.weighty = 1;
layout.setConstraints(resetButton, constraint);

JPanel panel = new JPanel(layout);
panel.add(slider);
panel.add(spinner);
panel.add(resetButton);
```

2.3 Hard to misuse

A well-designed API makes it easier to write correct code than incorrect code, and encourages good programming practices. It does not needlessly force the user to call methods in a strict order or to be aware of implicit side effects or semantic oddities.

To illustrate this point, let us temporarily leave the world of APIs to consider the syntax of HTML, plain TeX, and LaTeX. The table below shows a short example using all three markup languages:

Result:	the **goto label** statement
HTML:	`the <b>goto <u>label</u></b> statement`
Plain TeX:	`the {\bf goto \underline{label}} statement`
LaTeX:	`the \textbf{goto \underline{label}} statement`

Because HTML forces us to repeat the name of the tag in the close tag, it encourages us to write things like

```
the <b>goto <u>label</b></u> statement
```

which is badly nested and illegal. Plain TeX doesn't suffer from this problem, but it has its own flaws. Many TeX users, including the author of this document, often find themselves typing things like

```
the \bf{goto \underline{label}} statement
```

which results in "the **goto label statement**", i.e., the word "statement", and everything that follows until the next right brace, is in boldface. (This is because the `\bf` command applies to the current scope, while `\underline`'s

effect is restricted to its `{}`-delimited argument.) Another plain TeX oddity is that bold and italic are mutually exclusive. Thus,

```
the {\bf goto {\it label\/}} statement
```

produces "the **goto** *label* statement". A third quirk is that we must remember to insert the italic correction kerning (`\/`) after the italicized text. LaTeX solves all these problems, at the cost of introducing longer command names (`\textbf`, `\textit`, etc.).[5]

API design is, in many ways, like markup language design. An API is a language, or rather an extension to the programming language. The C++ code below mirrors the HTML example and its flaws:

```
stream.writeCharacters("the ");
stream.writeStartElement("b");
stream.writeCharacters("goto ");
stream.writeStartElement("i");
stream.writeCharacters("label");
stream.writeEndElement("i");
stream.writeEndElement("b");
stream.writeCharacters(" statement");
```

Wouldn't it be nice if we could let the C++ compiler detect bad nesting? Using the power of expressions, this can be done as follows:

```
stream.write(Text("the ")
             + Element("b",
                       Text("goto ")
                       + Element("u", "label"))
             + Text(" statement"));
```

Although the above code snippets are concerned with markup text generation, the same issues occur in other contexts, say, configuration files, where settings may be nested in groups:

```
QSettings settings;
settings.beginGroup("mainwindow");
settings.setValue("size", win->size());
settings.setValue("fullScreen", win->isFullScreen());
settings.endGroup();
```

There is another lesson to be drawn from markup languages, specifically HTML. The authors of HTML would like us to use `<em>` (emphasis)

[5] A reader commented: "This example is odd. I actually find HTML easier to read than TeX/LaTeX." Maybe so. But this section is titled "Hard to misuse", not "Leads to readable code".

instead of <i> and <strong> instead of <b>. However, the frowned-upon <i> and <b> tags are more convenient to type, and have a clearer semantics (does emphasis mean italic or bold?). If they had been serious about it, they would have replaced <i> with <font style="italic"> and likewise for bold, or omitted the tags altogether.

Qt! A common trap in Qt 3 was to swap the initial text and the parent of a `QPushButton`, `QLabel`, or `QLineEdit` when creating the object. For example:

```
button = new QPushButton(this, "Hello world");
```

The code compiled, but the widget didn't show any text, because "Hello world" was taken to be the object's name. In Qt 4, this is caught at compile-time. If you want to specify an object name, you must write

```
button = new QPushButton(this);
button->setObjectName("Hello world");
```

which is very explicit about your intentions.

Finally, we can often make an API hard to misuse by eliminating redundancy. For example, an `addItem(Item)` method that lets the user write

```
obj.addItem(yksi);
obj.addItem(kaksi);
obj.addItem(kolme);
```

is harder to misuse than the more general `insertItem(int,Item)` method, which encourages off-by-one errors:

```
// WRONG
obj.insertItem(0, yksi);
obj.insertItem(1, kaksi);
obj.insertItem(3, kolme);
```

2.4 Easy to extend

Libraries grow over time. New classes appear. Existing classes get new methods, methods get new parameters, and enums get new enum values. APIs should be designed with this in mind. The larger the API, the more likely the clashes between the new concepts and the existing concepts. In addition, binary compatibility concerns must be taken into consideration during the original design phase.

Qt! `QStyle` is a good example of a class that was difficult to extend in a binary compatible way in Qt 2. It had a whole series of virtual functions for painting the different widgets in Qt, which made it impossible to style new widgets (e.g., `QToolBox`) without breaking binary compatibility. In Qt 3,

QStyle's API was redesigned to be enum-based, so that new widgets could be supported by adding enum values.

2.5 Complete

Ideally, an API should be complete and let users do everything they want. It is rarely (if ever) possible to provide a complete API for anything, but at least it should be possible for users to extend or customize the existing API (e.g., by subclassing).

Completeness is also something that can appear over time, by incrementally adding functionality to existing APIs. However, it usually helps even in those cases to have a clear idea of the direction for future developments, so that each addition is a step in the right direction.

Chapter 3

The Design Process

APIs are usually the result of a design process that spans several years and involves many people. Every step in the process offers the opportunity to improve the API—or to spoil it. This chapter provides some guidance for designing a new API or extending an existing API. By following these guidelines, you increase your chances that the API will be a success rather than a liability.

3.1 Know the requirements

Before you set out to design or implement an API, you should have a good idea of the requirements. Sometimes, the requirements are reasonably clear (e.g., "implement POSIX"), but in most cases you will need to conduct some sort of requirements analysis. A good starting point is to ask as many people as possible (notably your boss, your colleagues, and potential users) about what features they would like to see.

Qt! When designing the new MDI classes for Qt 4.3, the main implementor wrote an email to the internal developers' mailing list where he asked about known issues with the previous MDI framework and any wishes for the new framework. Beyond helping define the API and feature set, this ensured that the other developers had their say and prevented dissension later on.

3.2 Write use cases before you write any other code

A common mistake when writing library components is to start by implementing the functionality, then design the API, and finally release it. APIs designed this way tend to reflect the structure of the underlying code, rather than what the application programmer who will use the API wants

to write. The implementation should adapt to the user, not the other way around.

Before you code an implementation or even design an API, start by writing a few typical application code snippets based on your requirement list. Don't worry at this stage about difficulties that would arise in implementing the API. As you write the code snippets, the API takes shape. The snippets should reflect the Perl motto "Make easy things easy and hard things possible".

Qt! For `QWizard`, the two main scenarios identified were linear wizards and complex wizards. In a linear wizard, the pages are shown sequentially, and no pages are ever skipped; in a complex wizard, different traversal paths are possible depending on user input, describing a directed acyclic graph. While the linear case can be seen as a special case of the complex case, it is common enough to deserve particular consideration.

3.3 Look for similar APIs in the same library

When designing a class called `XmlQuery`, look at the existing `SqlQuery` from the same class library. Chances are that they have similar concepts—you execute or precompile a query, navigate the result set, bind values to placeholders, and so on. Users who already know `SqlQuery` can then learn `XmlQuery` with very little effort. Furthermore, by mimicking an existing API, you implicitly benefit from all the incremental design work that went into that API and the feedback it received.

Of course, it would be foolish to blindly follow an existing API simply because it is there. Most existing APIs could do with some improvements, and being entirely consistent with them could turn out to be a two-edged sword. This having been said, an excellent way to achieve consistency with a bad API is to fix that API once and for all, and then mimic it.

Finding similar APIs in a large library is not always as easy as in the `XmlQuery/SqlQuery` case above. It requires patience, curiosity, and a powerful search tool such as `grep`.

If you are writing a new version of an API, you should know the API you are replacing like the back of your hand—otherwise, you will end up substituting new design flaws for old flaws. It might be tempting to believe that the old API is "utter crap" and doesn't deserve your time, but it is unlikely that it was all wrong, so try to take any good ideas on board. Moreover, since users will need to port code written for the old API, avoid gratuitous changes and support a superset of the old functionality.

Qt! Examples of dubious changes in Qt 4.0: `QDockWindow` was renamed `QDockWidget` (even though everybody says "dock window") and `QTextEdit::setOverwriteMode()` was removed, only to reappear in Qt 4.1.

3.4 Define the API before you implement it

For the same reason that you should write code snippets against an API before you define the API, you should specify the API and its semantics before you implement it. For a library with thousands of users, it is much better if the implementation is tricky and the API is straightforward than than the other way around.

Qt! Qt 4 lets the user specify the parent of a `QWidget` at any point by calling `setParent()`. In Qt 4.0 and 4.1, creating a parentless widget resulted in the creation of a window behind the scenes, which was very expensive. Delayed window creation, as introduced in Qt 4.2, is a beautiful example of an API-driven change to the implementation.

In contrast, the Qt 3 method `QWidget::recreate()` embodies the evils of implementation-driven APIs. Granted, on most platforms, `QWidget` created a native widget or window and wrapped it, but this implementation detail never belonged in the API.

The API and its semantics are the primary product that a library delivers. Experience has shown time and time again that APIs outlast their implementations (e.g., UNIX/POSIX, OpenGL, and `QFileDialog`).

As you implement the API or write unit tests for your implementation, you will most probably find flaws or undefined corner cases in your original design. Use this opportunity to refine your design, but don't let implementation considerations leak into the API, apart in exceptional cases (e.g., optimization options).

Qt! `QGraphicsScene::setBspTreeDepth()` probably constitutes such an exception. This method lets the user control the depth of the binary space partitioning (BSP) tree used to store the items of a graphics scene. `QGraphicsScene` tries to deduce an appropriate tree depth, but if the scene changes frequently, it may be more efficient to fix the tree depth. The name of the method is sufficiently intimidating to the user (who might not even know or care that `QGraphicsView` uses a BSP tree internally), suggesting that this is an advanced method.

3.5 Have your peers review your API

Look for feedback. Ask for feedback. Beg for feedback. Show your APIs to your peers, and collect all the feedback you get. Try to momentarily forget how much work it would be to implement the requested changes.

When receiving negative feedback, trust the principle that all feedback is useful. Any opinion (e.g., "This whole thing sucks", from Joe Blow) is a new piece of information, which you can recast into a fact and add to your mental list of facts (e.g., "Joe Blow, a Linux fanatic, strongly dislikes this

COM-inspired architecture”). The more facts you possess, the better the chances that you will design a good API.

3.6 Write several examples against the API

After you designed an API, you should write a few examples that use the API. Often, you can obtain examples simply by fleshing out the use cases defined earlier. It will also help if other people write examples using the API and give you their feedback.

Qt! The code snippets that were written initially when experimenting with the `QWizard` API eventually became the Class Wizard and the License Wizard examples provided with Qt, covering among them the linear and complex cases of a wizard. There is also a Trivial Wizard example, which involves no subclassing and has a “hello world” flavor.

3.7 Prepare for extensions

Expect your API to be extended in two ways: by the maintainers of the API, who will add to it (and occasionally deprecate parts of it), and by the users, who will write subclasses to customize the behavior of its components.

Planning for extensibility requires going through realistic scenarios and thinking of possible solutions. For each class that declares virtual methods, you should write at least three subclasses (as examples or as public classes in the API) to ensure that it is powerful enough to support a wide range of user requirements. This is sometimes called the rule of three.[1]

Qt! Qt 4.0 introduced `QAbstractSocket`, together with the concrete subclasses `QTcpSocket` and `QUdpSocket`. When adding `QSslSocket` in Qt 4.3, we were forced to add downcasts in some of `QAbstractSocket`’s methods, which should have been virtual but were not. “Manual polymorphism” worked in this case because `QSslSocket` lives in the same library as `QAbstractSocket`, but it wouldn’t have worked for a third-party component.

3.8 Don’t publish internal APIs without review

Some APIs start their glorious lives as internal APIs before they are made public. A common mistake is to forget to review them properly before taking this step. This story from Raymond Chen[2] illustrates the point:

[1] Will Tracz, “Software Reuse”, 1995. Available at `http://webcem01.cem.itesm.mx:8005/web/200111/cb00894/software_reuse.html`.

[2] Raymond Chen, “Why is the function SHStripMneumonic misspelled?”, The Old New Thing blog, 2008. Available at `http://blogs.msdn.com/oldnewthing/archive/2008/05/19/8518565.aspx`.

> If you wander through MSDN, you may stumble across the function `SHStripMneumonic`. The correct spelling is *mnemonic*. Why is the function name misspelled?
>
> "It was like that when I got here."
>
> The function was originally written for internal use only, and the person who wrote the function spelled the word incorrectly. Still, since it was an internal function, there was no real urgency to fix it.
>
> [...]
>
> In 2001, the order came down to document all functions which met specific criteria (the details of which I will not go into, and I would appreciate it if others didn't try), and the `SHStrip-Mneumonic` function was suddenly thrust onto the public stage before it had a chance to so much as comb its hair and make sure it didn't have food stuck in its teeth. The function had to be documented, warts and all, and the bad spelling was one of the warts.
>
> Of course, now that the function has been published, its name is locked and can't be changed, because that would break all the programs that used the original bad spelling.

3.9 When in doubt, leave it out

If you have your doubts about an API or functionality, you can leave it out, or mark it as internal, and reconsider it at a later point.

It often helps to wait for feedback from users. On the other hand, it is practically impossible to add every feature suggested by users. A good rule of thumb is to wait until at least three users have independently asked for a feature before implementing it.

Chapter 4

Design Guidelines

The guidelines presented in this chapter aim to help you get your APIs right. You can refer back to them at various stages of the process highlighted in the previous chapter. These guidelines were discovered by seeing them broken in actual APIs developed at Trolltech or elsewhere.

For many guidelines, we could formulate an equally true (but less often violated) counter-guideline. For example, from guideline 4.8, "Avoid making your APIs overly clever", we get the counter-guideline "Avoid making your APIs overly dumb", and from guideline 4.9, "Pay attention to edge cases", we get the counter-guideline "Focus on the general case". The need to consider many conflicting requirements is precisely what makes API design so rewarding. In the end, guidelines are no substitute for using your brain.

Naming

4.1 Choose self-explanatory names and signatures

To facilitate reading and writing code, pick names and signatures that are self-explanatory. The names should read like English.

Qt! `QPainterPath`'s main author recommended writing "vector path" rather than "painter path" in the documentation, because "everybody else calls it a vector path". Why doesn't Qt call it `QVectorPath` then?

Until Qt 4.2, another example was `QWorkspace`, which implemented MDI (multiple document interface). Why didn't MDI appear in the name? Thankfully, the new class that supersedes it is called `QMdiArea`.

The meaning of the arguments to a method should be clear from the context at the call site. Beware of `bool` parameters, which often lead to unreadable code.

Qt! `QWidget::repaint()` takes an optional `bool` that controls whether the widget should be erased before it is repainted. Code like `repaint(false)` leaves some readers wondering whether the method does anything at all ("don't repaint!"). At the very least, the argument should have been an enum.

Strive for consistency in naming. If you use the terms "widgets" and "controls" interchangeably, users will be mislead into believing that there is a difference. And if you use the same term for two distinct concepts, users will be even more confused, until they read the documentation. Consistency is particularly important when fixing the order of parameters. If rectangles are described as $(x, y, width, height)$ in one part of the API, they shouldn't become $(x, width, y, height)$ elsewhere.

Qt! In the run-up to Qt 4.0, `QStackArray` implemented a vector stored on the call stack, much like the C99 variable-length arrays (VLAs). What was wrong with it? Qt already had a `QStack` container class, implementing a FIFO (first in, first out) list. So `QStackArray` sounded like an array of FIFO lists to anyone who knew about `QStack`. To avoid overloading the word "stack", we renamed the class `QVarLengthArray`.

Good naming also requires knowing the audience. If you define a powerful API for XML, you will most certainly need to use XML jargon in the API names. (You should also define it in the documentation and provide links into the relevant standards.) If, on the contrary, you are designing a high-level API for XML, don't expect your readers to know what IDREFs and NCNames are.

Giving good parameter names is also part of designing an API, even in languages like C++ where they are not part of the signature. Parameter names figure both in the documentation and in the autocompletion tooltips provided by some development environments. Renaming poorly-named parameters is tedious and error-prone, so give the same care to parameter naming as you give to the other aspects of the API. In particular, avoid all single-letter parameter names.[1]

4.2 Choose unambiguous names for related things

If two or more similar concepts need to be distinguished, choose names that map unambiguously to the concepts they denote. More specifically, given the set of names $\{n_1, \ldots, n_k\}$ and the set of concepts $\{c_1, \ldots, c_k\}$, it should be easy for anyone to associate the names with the right concepts without looking them up in the documentation.

[1]Exceptionally, the names `x`, `y`, and `z` for plane coordinates are invariably superior to their pedantic cousins `abscissa`, `ordinate`, and `applicate`.

Suppose you have two event delivery mechanisms, one for immediate (synchronous) delivery and one for delayed (asynchronous) delivery. The names `sendEventNow()` and `sendEventLater()` suggest themselves. If the intended audience is expected to know the "synchronous" terminology, you could also call them `sendEventSynchronously()` and `sendEventAsynchronously()`.

Now, if you want to encourage your users to use synchronous delivery (e.g., because it is more lightweight), you could name the synchronous method `sendEvent()` and keep `sendEventLater()` for the asynchronous case. If you would rather have them use asynchronous delivery (e.g., because it merges consecutive events of the same type), you could name the synchronous method `sendEventNow()` and the asynchronous method `sendEvent()` or `postEvent()`.

Qt! In Qt, `sendEvent()` is synchronous, and `postEvent()` is asynchronous. One way to keep them separated is to observe that sending an email is usually much faster than posting a letter using regular mail. Still, the names could have been clearer.

A similar principle applies when naming the parameter of a copy constructor or of an assignment operator in C++. Because it offers good contrast with the implicit `this` parameter, `other` is a good name. A poorer choice would be to call it after the class:

```
// SUBOPTIMAL
Car &Car::operator=(const Car &car)
{
    m_model = car.m_model;
    m_year = car.m_year;
    m_mileage = car.m_mileage;
    ...
    return *this;
}
```

What is wrong with this? This method deals with two `Car`s: `*this` and `car`. The name `car` is appropriate in contexts where there is only one `Car`, but it is not helping much when there are two or more.

4.3 Beware of false consistency

APIs, like good writing, should display symmetry. In his American English style book[2], William Strunk Jr. enjoins us to "express co-ordinate ideas in

[2] William Strunk Jr., *The Elements of Style*, 1918. Available online at `http://www.bartleby.com/141/`.

similar form", adding:

> This principle, that of parallel construction, requires that expressions of similar content and function should be outwardly similar. The likeness of form enables the reader to recognize more readily the likeness of content and function. Familiar instances from the Bible are the Ten Commandments, the Beatitudes, and the petitions of the Lord's Prayer.

The flip side of this rule is that asymmetry of function should be reflected by asymmetry of form. Consequently, if you make a habit of prefixing setter methods with "set" (e.g., `setText()`), then avoid that prefix for methods that aren't setters.

Qt! This guideline has been violated every so often in Qt's history. The Qt 3 method `QStatusBar::message(`*`text`*`, `*`msecs`*`)` displayed a message in the status bar for a specified amount of milliseconds. The name looks like that of a getter, even though the method is non-`const` and returns `void`. For Qt 4, we considered renaming it `setMessage()` for "consistency". However, with its two parameters, the method doesn't feel quite like a setter. In the end, we settled for the name `showMessage()`, which doesn't suggest false analogies.

The event delivery example from the previous guideline can also serve as illustration here. In C++, since a synchronous mechanism delivers the event immediately, the event can usually be created on the stack and passed as a `const` reference, as follows:

```
KeyEvent event(KeyPress, key, state);
sendEventNow(event);
```

In contrast, the asynchronous mechanism requires creating the object using `new`, and the object is deleted after it has been delivered:

```
KeyEvent *event = new KeyEvent(KeyPress, key, state);
sendEventLater(event);
```

Thanks to static typing, the compiler will catch the following bad idioms:

```
// WON'T COMPILE
KeyEvent *event = new KeyEvent(KeyPress, key, state);
sendEventNow(event);

// WON'T COMPILE
KeyEvent event(KeyPress, key, state);
sendEventLater(event);
```

False consistency would suggest making the two methods' signatures identical, despite the asymmetry of function (namely, `sendEventLater()` takes ownership of the event, whereas `sendEventNow()` doesn't).

Qt! In Qt, we walked straight into the trap: `sendEvent()` and `postEvent()` both take a pointer, which makes it very easy to leak memory or double-delete the event.

From an API perspective, a better solution would have been to restore the symmetry by having `sendEventNow()` take ownership of the event or `sendEventLater()` copy it. However, this is less efficient. To attract the favor of the gods, we must routinely sacrifice pure API design principles on the altars of performance and ease of implementation.

4.4 Avoid abbreviations

In an API, abbreviations mean that users must remember which words are abbreviated, and in which contexts. For this reason, they should generally be avoided. An exception is often made for very common, unambiguous forms such as "min", "max", "dir", "rect", and "prev", but then this convention must be applied consistently; otherwise, users will inadvertently invoke `setMaximumCost()` instead of `setMaxCost()`, and the day after they will try to call `setMaxWidth()` instead of `setMaximumWidth()`. Although errors like this are caught by the compiler, they can aggravate users.

Qt! The examples above come straight from Qt 4 (`QCache` and `QWidget`). Similarly, Qt 4 has `QDir` and `QFileInfo::isDir()` but also `QFileDialog::setDirectory()` and `QProcess::workingDirectory()`.

This guideline applies to abbreviations, not to acronyms. To separate the two, listen in the hallway: XML is called XML, not Extensible Markup Language, so it qualifies as an acronym; but JavaScript is JavaScript, not JS.

In addition, since users don't have to type parameter names, abbreviations are acceptable in that context, so long as their meaning is clear.

4.5 Prefer specific names to general names

Prefer specific names to general ones, even if they sometimes seem too restrictive. Remember that names occupy API real estate: Once a name is taken, it cannot be used for something else. This is especially an issue with class names, since a library might contain hundreds or even thousands of classes that need distinct names. Another reason for preferring more specific names is that they are easier for users to relate to.

Qt! The `QRegExp` class provides support for regular expressions, but also for wildcards and fixed strings. It could also have been called `QStringPattern`, but this more general name hides what the class is used for in the vast majority of cases, namely, regular expressions. For this reason, the name `QRegExp` is arguably the most appropriate for the class.

If you need an error-reporting mechanism for SQL, call the abstract base class `SqlErrorHandler` rather than just `ErrorHandler`, even if the implementation is not tied to SQL. Arguably, calling the class `ErrorHandler` could facilitate its reuse in other parts of the API, but in practice this kind of reuse is unlikely, because it is hard to foresee future requirements. Someday, you might need an error handling mechanism for XML that is slightly different from the SQL error handing and call it `XmlErrorHandler`. Ten years down the road, you might finally discover a way to unify all error handlers—then, you will be thankful that the name `ErrorHandler` is still available, and use it as a base class for all the specific handler classes in the next version of the library, when you can break binary compatibility.

Qt! For many years, Qt provided two APIs for XML: SAX and DOM. A nice, symmetric naming convention would have been to use `QSax` and `QDom` as the class prefixes, but unfortunately we settled for `QXml` for SAX and `QDom` for DOM. The asymmetry, annoying on esthetic grounds, became plain confusing as we introduced new XML classes in Qt 4.3 and 4.4, also with the `QXml` prefix. Looking at the class names, nobody call tell that `QXmlReader` and `QXmlSimpleReader` are part of the now semi-deprecated SAX API, whereas `QXmlStreamReader` is the preferred Qt approach to parse XML documents.

4.6 Don't be a slave of an underlying API's naming practices

A common requirement is to wrap an existing API or implement a file format or protocol. These normally come with their own names, which might be poorly chosen or clash with the names in your library. As a general rule, don't hesitate to invent your own terminology. For example, if you to wrap the OpenGL `pbuffer` extension in a class, call it `GLPixelBuffer`, not `GLPbuffer`.

Similarly, when porting an API to a new programming language, you should try to merge it well with your library, rather than subject yourself to the foreign conventions. The few users who know the existing API in the other language won't mind the improvements, whereas the remaining users will greatly benefit from them.

Qt! The SAX classes in Qt are modeled after the de facto standard SAX API for Java. Some changes were made to accommodate C++ and Qt, but on the whole the C++ architecture is very similar to the Java one—much too

similar, in fact. For example, the XML data is provided by a class called QXmlInputSource, which can be initialized with a QIODevice; a tighter integration with Qt would have used QIODevice directly and eliminated QXmlInputSource.

Semantics

4.7 Choose good defaults

Choose suitable defaults so that users won't need to write or copy and paste boilerplate code just to get started. For example, in user interface design, a good default is to respect the platform's native look and feel. Using Qt for Mac OS X, we simply write

```
QPushButton *button = new QPushButton(text, parent);
```

and we obtain a native-looking Aqua push button with the specified text and an appropriate preferred size (its "size hint"), ready to be inserted in a layout manager. Contrast this with the following code, written using Apple's Cocoa framework:

```
static const NSRect dummyRect = {
    { 0.0, 0.0 }, { 0.0, 0.0 }
};
NSButton *button = [[NSButton alloc]
                            initWithFrame:dummyRect];
[button setButtonType:NSMomentaryLightButton];
[button setBezelStyle:NSRoundedBezelStyle];
[button setTitle:[NSString stringWithUTF8String:text]];
[parent addSubview:button];
```

Why do we need to set the button type and the bezel style to get a plain Aqua button? If compatibility with NeXTstep and OpenStep is that important, why not provide a convenience class (NSPushButton or NSRoundedButton or NSAquaButton, whatever) with the right defaults? And why do we need to specify a position and a size to the initializer method, when we probably need to set them later anyway based on the window size and the button's reported preferred size?

But choosing good defaults isn't just about eliminating boilerplate code. It also helps make the API simple and predictable. This applies especially to Boolean options. In general, name them so that they are false by default and enabled by the user. In some cases, this requires prefixing them with No or Dont or Anti. This is acceptable, although alternatives should be sought

when possible (e.g., `CaseInsensitive` rather than `NotCaseSensitive`). In all cases, resist the temptation to enable all sorts of bells and whistles by default just to show off.

Finally, remember that choosing good defaults doesn't excuse you from your duty to document them.

4.8 Avoid making your APIs overly clever

We observed earlier that the semantics of an API should be simple and clear, and follow the principle of least surprise. If an API is too clever (e.g., if it has subtle side-effects), the users will not understand it as well and will find themselves with subtle bugs in their applications. Cleverness also contributes to making the code less readable and maintainable, and increases the need for documentation.

Qt! `QLabel` in Qt 3 offers a beautiful example of cleverness. The `setText()` method takes a string, which may be plain text or HTML. Qt automatically detects the format by inspecting the string. This is sometimes a bit too clever and occasionally results in the wrong format being chosen (which is a real problem if the string comes from the end-user). On the other hand, the automatic detection usually works, is very convenient, and can be overridden by calling `setTextFormat()`, so one could argue that this acceptable.

On the other hand, calling `setText()` with an HTML string in Qt 3 also had the side-effect of setting the `wordWrap` property to `true` (the default being `false`). In turn, word-wrapping enables height-for-width, a layout feature that impacts the entire dialog. As a result, users who changed "Name" to "Name" in one label found themselves with a totally different dialog resizing behavior and couldn't understand why.

4.9 Pay attention to edge cases

In the context of library programming, the importance of getting edge cases right cannot be overstated. If you hear a colleague in the hallway say "It doesn't matter, it's just a corner case", please slap them in the face.

Edge cases are critical because they tend to ripple through an API. If your fundamental string-search algorithm has bad edge-case semantics, this might lead to bugs in the regular expression engine, which would lead to bugs in applications that use regular expressions. But edge cases are also important in their own right: Nobody is interested in a factorial function that computes $0! = 0$.

Qt! The Qt 3 static method `QStringList::split(`*`separator`*`, `*`string`*`, `*`allowEmptyEntries`* `= false)` split *`string`* at every occurrence of *`separator`*. Thus, `split(':', "a:b:c")` returned the three-item list [`"a", "b", "c"`]. One could have expected that if *`separator`* occurs n times in *`string`*, the

resulting list would contain $n + 1$ items. However, `split()` managed to violate this rule in two ways. First, the *`allowEmptyEntries`* parameter defaulted to `false`. Second, as a special case, the method always returned the empty list if *`string`* was empty, rather than the one-item list [`""`]. Both issues lead to subtle bugs in applications and bug reports to Trolltech.

In Qt 4, the method has been replaced by `QString::split(`*`separator`*`,` *`behavior`* `= KeepEmptyParts,` *`caseSensitivity`* `= Qt::CaseSensitive)`, which doesn't suffer from these flaws.

If the abstractions upon which you build handle edge cases gracefully, your code is most likely to do so as well. To mismanage edge cases, you often need extra code. Consider the following `product()` function:

```
double product(const QList<double> &factors)
{
    // WRONG
    if (factors.isEmpty())
        return 0.0;

    double result = 1.0;
    for (int i = 0; i < factors.count(); ++i)
        result *= factors[i];
    return result;
}
```

If passed an empty list, the function returns 0, which is mathematically unsound: The neutral element of multiplication is 1, not 0. We could of course fix the check at the beginning of the function:

```
    if (factors.isEmpty())
        return 1.0;
```

But is this check necessary in the first place? It turns out it isn't. The rest of the function relies on C++'s `for` loop and multiplication operator, which handle edge cases gracefully. The emptiness check is a false optimization—it barely optimizes the trivial case, which is already blindingly fast, while slowing down all other cases.

When implementing an API, start by handling the general case without worrying about edge cases, and then test the edge cases. More often than not, you will find that you don't need any extra code for handling the edge cases. (And when you do, it is usually a sign that one of the underlying programming constructs is to blame.) Make sure, though, that the edge cases are properly covered by unit tests, so that their semantics doesn't change between releases as a result of optimizations or other changes to the code.

4.10 Be careful when defining virtual APIs

Virtual APIs are notoriously difficult to get right and can easily break between releases. This is often called the "fragile base class problem". When designing a virtual API, there are two main mistakes to avoid.

The first mistake to define too few virtual methods. For library authors, it is hard to foresee all the ways in which the library will be used. By omitting the `virtual` keyword in front of a method declaration, we often reduce the reusability and customizability of the entire class.

The second mistake is to make every method virtual. In a language like C++, this is inefficient, but more importantly, it delivers a false promise. More often than not, it would be useless, if not dangerous, to reimplement some methods, because the class's implementation doesn't call them or expects them to have a specific semantics.

The proper approach requires thinking through what parts of the API should be virtual and which parts should be non-virtual, and specifying clearly in the documentation how the class uses the method itself.

Qt! The Spreadsheet example in the official Qt 3 book[3] reimplemented `QTableItem::text()`, a virtual method. The example worked fine against Qt 3.2.0, but was broken in a later Qt 3 release because some calls to `text()` in `QTable` were replaced with direct access to the member variable that stores the text in the default `text()` implementation, bypassing any subclass.

It usually pays off to cleanly separate a class's public API, which is used directly by applications, and the virtual API, which can be reimplemented to customize the behavior of the class. In C++, this means making all virtual methods protected.

Qt! In Qt 4, `QIODevice` successfully follows this approach. Direct users of the class call `read()` and `write()`, while subclass authors reimplement `readData()` and `writeData()`. In early implementations of `QIODevice`, the public, non-virtual methods simply called the protected, virtual methods. Since then, we added buffering, newline conversions, an "unget" buffer, and a quick code path for reading one character at a time, all of this in the non-virtual `read()` and `write()` methods.

In a way, `QWidget` follows this pattern as well with its event delivery mechanism. Direct users of the class call `show()`, `resize()`, or `repaint()`, whereas subclass authors reimplement `showEvent()`, `resizeEvent()`, or `paintEvent()`.

An annoying limitation of C++ is that we cannot add new virtual methods to an existing class without breaking binary compatibility. As an ugly

[3] Jasmin Blanchette and Mark Summerfield, *C++ GUI Programming with Qt 3*, Prentice Hall, 2004. Available at `http://www.informit.com/content/images/0131240722/downloads/blanchette_book.pdf`.

work around, get into the habit of providing a general virtual method that can be used later on if you need to extend the class. For example:

```
virtual void virtual_hook(int id, void *data);
```

Structural

4.11 Strive for property-based APIs

Some APIs force users to specify all the attributes of an object upfront at creation-time. Many low-level platform-specific APIs work this way. The following example comes straight from a Win32 application:

```
m_hWindow =
  ::CreateWindow("AppWindow",    /* class name */
                m_pszTitle,     /* title to window */
                WS_OVERLAPPEDWINDOW, /* style */
                CW_USEDEFAULT,  /* start pos x */
                CW_USEDEFAULT,  /* start pos y */
                m_nWidth,       /* width */
                m_nHeight,      /* height */
                NULL,           /* parent HWND */
                NULL,           /* menu HANDLE */
                hInstance,      /* */
                NULL);        /* creatstruct param */
```

The need to display each argument on its own line and to accompany it with a comment to increase readability is usually an indication that a procedure has too many parameters.

In contrast, a property-based approach lets users create an object without specifying any arguments to the constructor. Instead, users can set the properties that define the object one by one, in any order. For example:

```
window = new Window;
window->setClassName("AppWindow");
window->setWindowTitle(winTitle);
window->setStyle(Window::Overlapped);
window->setSize(width, height);
window->setModuleHandle(moduleHandle);
```

To the users, there are many advantages to this approach:

- It offers a very intuitive model.

- Users don't need to remember in which order they need to supply the attributes or options.
- User code is more readable and doesn't require additional comments (which may be out of sync with the code).
- Since properties have default values, users only have to set those that they explicitly want to change (e.g., in the window example above, they don't have to set the x and y coordinates, the parent window handle, the menu handle, or the "creatstruct"—whatever that is).
- Users can change the value of a property at any time, instead of having to replace the object with a new one whenever they want to modify it.
- By calling getters, users can query back everything they set, which helps debugging and is useful in some applications.
- The approach is compatible with graphical property editors, which let the user see the result of setting a property immediately.

To the library developers, designing property-based APIs requires more thinking. Since the properties can be set in any order, it is usually necessary to use lazy initialization and other techniques in the implementation to avoid recomputing the whole object every time a property changes.

Qt! Take `QRegExp`. Users can set it up in one go if they want:

```
QRegExp regExp("*.wk?", Qt::CaseInsensitive,
               QRegExp::Wildcard);
```

But they can also initialize it step by step:

```
QRegExp regExp;
regExp.setPattern("*.wk?");
regExp.setCaseSensitivity(Qt::CaseInsensitive);
regExp.setPatternSyntax(QRegExp::Wildcard);
```

The `QRegExp` implementation delays the compilation the regular expression or wildcard pattern to the point when the `QRegExp` object is actually used to match some text.

4.12 The best API is no API

In a movie, the best special effects are those that you don't notice. A similar principle applies to API design: The ideal features are those that require no (or very little) additional code from the application writer.

Qt! In Qt 3, widgets were restricted to 32 767 × 32 767 pixels on most platforms. To render scenes larger than that in a `QScrollView` (e.g., large web pages), the only solution was to subclass `QScrollView` and reimplement `drawContents()`, `contentsMousePressEvents()`, etc. Qt 4 works around the widget size limitation internally for all `QWidget`s. As a result, we can now draw large scenes in a `QWidget`'s `paintEvent()` the standard way and insert the widget directly into a `QScrollArea`, eliminating the need for methods like `drawContents()` and `contentsMousePressEvents()`.

Other examples of significant Qt 4 features that have little API are PDF support (which is enabled by calling `setOutputFormat(QPrinter::PdfFormat)` on a `QPrinter`), style sheets (which are set by calling `setStyleSheet()`), and 2D painting using `QPainter` on an OpenGL widget (which just works).